U12 12. Übung

U12-1 Überblick über die 12. Übung

- Besprechung 8. Aufgabe (buffer)
- Besprechung der 2. Miniklausur
- PV-chunk Semaphore
- Semaphore vs. Mutex- und Condition-Variablen

U12-2 PV-chunk Semaphore

- Erweiterung des Konzepts der zählenden Semaphore
  - Semaphor **s** wird im Rahmen der P- und V-Operation nicht nur um 1 dekrementiert bzw. inkrementiert, sondern um einen Wert **n**
  - P-Operation blockiert, falls `s - n < 0`
- ermöglicht eine einfaches Warten, bis eine bestimmte Anzahl eines Betriebsmittels verfügbar ist
  - in der Aufgabe 8: Warten bis **n** Pufferplätze gefüllt sind

U12-3 Semaphore vs. Mutexes und Conditions

- Semaphore sind der "abstraktere" Koordinierungsmechanismus
- Vorteile:
  - Koordinierungscode in der Anwendung ist schlanker
  - Semantik von P- und V-Operationen ist allgemein bekannt, Koordinierung damit unmittelbar verständlich
  - Anwendungsprogramm ist besser lesbar, weil eigentliche Funktionalität nicht zu sehr durch Koordinierungscode unterbrochen ist
- Nachteile:
  - weniger Flexibilität als beim expliziten Umgang mit Mutex-Locks und Condition-Variablen
    - z. B. wenn man anwendungsabhängig entscheiden will, ob man aktiv wartet (Spin-Lock) oder den Thread schlafen legt (cond_wait)

U12-4 Lösungsskizze zum buffer-Modul

! Fehlerbehandlungen sind z. T. verkürzt oder vernachlässigt

1 Semaphor-Modul

- Semaphor-Datenstruktur

```
typedef struct {
    int s;      /* Zustand */
    pthread_mutex_t m;
    pthread_cond_t c;
} SEM;
```

1 … Semaphor-Modul

■ Initialisierungs-Funktion

```
SEM *sem_init(n) {
    SEM *s;
    if ((s = malloc(sizeof SEM)) == NULL) {
        return NULL;
    }
    s->s = n;
    pthread_mutex_init(&s->m, NULL);
    pthread_cond_init(&s->c, NULL);
    return s;
}
```

■ Lösch-Funktion

```
void sem_delete(SEM *s) {
    pthread_mutex_destroy(&s->m);
    pthread_cond_destroy(&s->c);
    free(s);
}
```


1 … Semaphor-Modul

■ P- Operation

```
void P(SEM *s, int n) {
    pthread_mutex_lock(&s->m);
    while (s->s < n) {
        pthread_cond_wait(&s->c, &s->m);
    }
    s->s -= n;
    pthread_mutex_unlock(&s->m);
}
```

■ V- Operation

```
void V(SEM *s, int n) {
    pthread_mutex_lock(&s->m);
    s->s += n;
    pthread_cond_broadcast(&s->c);
    pthread_mutex_unlock(&s->m);
}
```


2 jbuffer-Initialisierung

```
/* Grundidee: alle Puffer-relevanten Daten werden in einer
   Datenstruktur zusammengefasst, die dann an die Threads
   uebergeben wird.
   Dadurch kann man die jbuffer-Funktion ggf. auch mehrmals
   aufrufen und die Instanzen arbeiten unabhaengig voneinander
*/

struct buffer {
    char *data;
    int size;
    int delay;
    int w_index;
    int r_index;
    int end;
    SEM *free_slots;
    SEM *filled_slots;
    FILE *in_stream;
    FILE *out_stream;
};
```


2 … jbuffer-Initialisierung

```
void jbuffer(int fd1, int fd2, int bufsize, int bufdelay) {
    pthread_t t1, t2;

    struct buffer b;
    /* b initialisieren */
    if ((b.data = malloc(bufsize)) == NULL) { ... }
    b.size = bufsize; b.delay = bufdelay;
    b.w_index = b.r_index = 0; b.end = -1;

    if ((b.free_slots = sem_init(bufsize)) == NULL) { ... }
    if ((b.filled_slots = sem_init(0)) == NULL) { ... }

    if ((b.in_stream = fdopen(fd1, "r")) == NULL) { ... }
    if ((b.out_stream = fdopen(fd2, "w")) == NULL) { ... }

    /* Threads erzeugen und b uebergeben */
    pthread_create(&t1, NULL, producer, &b);
    pthread_create(&t2, NULL, consumer, &b);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    /* abschliessend b.data, Semaphore und Streams freigeben */
    ...
}
```

3 Producer

```
static void *producer(void *arg) {
    struct buffer *b = arg;
    int c;

    while ((c = getc(b->in_stream) != EOF) {
        P(b->free_slots, 1);
        b->data[b->w_index] = c;
        b->w_index = (b->w_index + 1) % b->size;
        V(b->filled_slots, 1);
    }

    /* Dateiende-Markierung: buffer-Inhalt = 0 && end == index*/
    P(b->free_slots, 1);
    b->data[b->w_index] = 0;
    b->end = b->w_index;
    /* consumer auch deblockieren, falls er auf delay wartet */
    V(b->filled_slots, b->delay);
}
```


4 Consumer

```
static void *consumer(void *arg) {
    struct buffer *b = arg;
    int i;

    /* auf delay Zeichen warten und dann die Auswirkung von P
        rueckgaengig machen - keine schoene Loesung
        ein "nur testendes P" waere besser */
    P(b->filled_slots, b->delay);
    V(b->filled_slots, b->delay);

    while (1) {
        P(b->filled_slots, 1);
        /* EOF-Bedingung abpruefen */
        if ( b->data[b->r_index] == 0 && b->r_index == b->end )
            break;
        putc(b->data[b->r_index], b->out_stream);
        b->r_index = (b->r_index + 1) % b->size;
        V(b->free_slots, 1);
    }
}
```